VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

RYCHLÉ DOTAZOVÁNÍ NAD METADATY JAZYKA JAVA

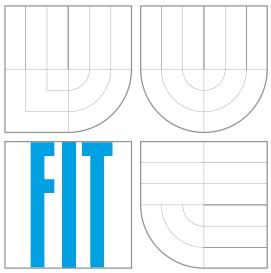
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VLADIMÍR FALTÝN

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

RYCHLÉ DOTAZOVÁNÍ NAD METADATY JAZYKA JAVA

FAST QUERIES OVER JAVA LANGUAGE METADATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VLADIMÍR FALTÝN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KŘIVKA ZBYNĚK, Ph.D.

BRNO 2015

Abstrakt

Cílem této bakalářské práce je navrhnout a vytvořit jazyk pro dotazování nad metadaty jazyka Java. Metadata získáme dekompilátorem Procyon. Získaná metadata uložíme do grafové databáze Titan. Pro práci s Titanem použijeme grafový framework TinkerPop. Pro dotazování do databáze použijeme navržený dotazovací jazyk. Překladač pro dotazovací jazyk vygenerujeme nástrojem ANTLR.

Abstract

The goal of this bachelor thesis is to design and implement a query language over metadata of Java source programs. The metadata is obtained by Procyon decompiler and stored in Titan graph database. To access Titan, TinkerPop framework is used. Then, the designed query language is used to acces the data in the database. The parser for the query language is generated using ANTLR tool.

Klíčová slova

dotaz, metadata, Java, jazyk, TinkerPop, Titan, Procyon, ANTLR

Keywords

query, metadata, Java, language, TinkerPop, Titan, Procyon, ANTLR

Citace

Vladimír Faltýn: Rychlé dotazování nad metadaty jazyka Java, bakalářská práce, Brno, FIT VUT v Brně, 2015

Rychlé dotazování nad metadaty jazyka Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl Ing. Ondřej Žižka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vladimír Faltýn
19. května 2015

Poděkování

Chtěl bych poděkovat panu Ing. Zbyňku Křivkovi, Ph.D. mému vedoucímu a zadavateli Ing. Ondřeji Žižkovi za trpělivost a užitečné rady, které mi pomohly při tvorbě této bakalářské práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Cílem této práce je navrhnout a implementovat jazyk pro dotazování nad metadaty jazyka Java, který bude schopen vyhledávat třídy v jar souborech podle zadaných kritérií. Aplikace bude sloužit pro migraci Java aplikací a jejích analýzu. Výsledná aplikace bude dostupná jako knihovna s množností i samostatného spuštění.

Druhá kapitola pojednává o specifikaci požadavků a jejích následné analýze. Nejprve popíšeme specifikaci dotazovacího jazyka a ukážeme pár příkladů, které ilustrují schopnosti jazyka. Poté vyhledáme vhodný typ databáze, který použijeme. Následně vybereme dekompilátor pro získání metadat. A nakonec popíšeme tvorbu překladače dotazovacího jazyka.

Ve třetí kapitole provedeme rešerši nástroje Procyon pro dekompilování a analýzu class souborů jazyka Java.

Čtvrtá kapitola popisuje návrh aplikace. Kde popíšeme jazyk pro dotazování, strukturu grafové databáze a návrh struktury aplikace.

Pátá kapitola se zabývá implementačními detaily. Kde popíšu API pro práci s knihovnou, způsob dekompilace jar souboru, tvorbu grafu a interpret dotazovacího jazyka.

Šestá kapitola popisuje průběh testování aplikace a demonstrujeme sílu navrženého jazyka. Testování provádíme na několika dotazech, kde popíšeme dotaz a jeho výsledek.

V sedmé kapitole uvedu nabyté poznatky, které jsem získal při práci na projektu. Zhodnotím vlastní přínos a naznačím směr, kterým by se mohla aplikace ubírat.

Kapitola 2

Specifikace a analýza požadavků

Nejprve se podíváme na specifikaci dotazovacího jazyka. Poté vybereme vhodnou databázi pro uložení dat z class souborů, do které se budeme dotazovat dotazovacím jazykem. Následně vyhledáme dekompilátor class souboru, ze kterého získáme potřebná data. Nakonec řekneme, jak budeme vytvářet překladač pro náš dotazovací jazyk.

2.1 Specifikace dotazovacího jazyka

Zde si rozebereme specifikaci kladenou na výsledný dotazovací jazyk. Specifikace byla doplňována postupně, jak se projekt vyvíjel. Jde spíše o experimentální projekt pro navržení dotazovacího jazyka, který bude vhodný pro získávání dat z již zkompilovaných java souborů, tedy class souborů.

Hlavním požadavkem je dotazování na metadata class souborů jazyka Java[1]. Z metadat nás zajímají informace ohledně názvů tříd, cesty, typ třídy (jestli se jedná o třídu, rozhraní, výčtový typ, anotaci nebo abstraktní třídu). Dále nás zajímají modifikátory třídy, vztahy mezi třídami (např. dědičnost tříd, implementace rozhraní, volání metod). A posledním požadavkem je informace o použití anotací a to především u tříd, metod a parametrů metod. Pro získání všech těchto informací nám metadata nebudou stačit, budeme potřebovat ještě čitelnou formu byte kódu. Proto si vyhledáme vhodný dekompilátor, který umožňuje získání metadat a byte kódu.

Dalším ze specifikovaných požadavků je dotazovat se na všechny třídy v jar souboru [1]. To znamená, že budeme muset pokládat dotazy i na velké množství tříd. Z toho důvodu se budou získaná data ukládat do databáze, než se na ně začneme dotazovat. Pro uložení takového množství data si zvolíme vhodný databázový systém.

Zde následují příklady slovních dotazů, které se budou pokládat. Nejčastěji se budeme ptát na názvy tříd a anotace.

- Chci všechny třídy, které rozšiřují nebo jsou importovány do třídy C a zároveň tyto třídy implementují rozhraní, které je rozšířeno o interface I.
- Chci všechny třídy, které implementují rozhraní I.
- Chci všechny třídy, které volají metodu z rozhraní I.
- Chci všechny třídy, které volají metodu z rozhraní, které je rozšířeno o rozhraní I.
- Chci všechny třídy, které volají metodu daného jména z rozhraní, které je rozšířeno o rozhraní I s danými parametry.

- Chci všechny třídy, které volají metodu z třídy, která implementuje rozhraní I a je anotovaná anotací @A.
- Chci všechny třídy, které volají metodu z třídy, která implementuje rozhraní I a je anotovaná anotací @A s parametrem @A(foo="bar").
- Chci všechny třídy, které jsou anotovány anotaci @A.
- Chci všechny třídy, které importují podtřídu z třídy B.
- Chci všechny třídy, které importují podtřídu z třídy, která implementuje podrozhraní z rozhraní I.

Důvodem, proč je zapotřebí jazyka pro dotazování nad class, je migrace Java aplikací, analýza class souborů a převod starých aplikací do novějších verzí Javy.

2.2 Výběr typu databáze

Specifikovali jsme si zadání a zjistili jsme, že potřebujeme najít vhodnou databázi pro ukládání získaných dat z class souborů. Hledáme databázi, která je schopná uložit velké množství dat a zároveň rychle a efektivně pracovat s daty. Dalším požadavkem je uložení pojmenovaných vztahů mezi záznamy s možností uložit vlastnosti k vztahům.

2.2.1 Relační databáze

Relační databáze (Relational Database Management System, RDBMS) je založena na tabulkách, kde řádky chápeme jako záznamy s primárním klíčem, který uchovává informaci o relaci mezi jednotlivými záznamy. Databáze odstraňuje nedostatky předešlých databází, splňuje podmínky ACID (atomičnost, konzistence, izolovanost, trvanlivost) a obsahuje databázové triggery. Nejznámějším představitelem této databáze je SQL od ORACLE.

Konstruktorem relačního modelu je relace (tabulky databáze), což jsou dvojrozměrné struktury tvořené záhlavím a tělem, kde jeden řádek v tabulce představuje záznam dat. A sloupce představují atributy, které mají určen svůj datový typ a doménu určující přípustnou množinu hodnot atributu.

Vlastnosti vztahu u relačního modelu jsou stupeň vztahu, kardinalita a volitelnost účasti. Stupně vztahu jsou následující unární (relace spojení sama se sebou), binární (vztah mezi dvěma relacemi), ternární (vztah tří relací najednou), n-ární (vztah n–relací najednou). Kardinalita mezi relacemi může být následující 1:1 (jednomu záznamu z jedné tabulky odpovídá právě jeden záznam z druhé tabulky), 1:N (jednomu záznamu z jedné tabulky odpovídá více záznamu z druhé tabulky), N:M (umožňuje přiřadit N záznamu z jedné tabulky k M záznamům z druhé tabulky). A poslední vlastností vztahu je volitelnost, zda je účast relace ve vztahu povinná či volitelná.

Relační databáze jsou v dnešní době nejpoužívanější, mají matematický podklad, jsou definované pomocí algebry. Zavedl se pojem ACID a databázové triggery. Pro naši databázi by to mohla být jedna z možností.

2.2.2 Objektově orientované databáze

Objektově orientované databáze (Object Database Management System, ODBMS) jsou systémy správy databází, ve kterých informace představuje objekt stejně jako v objektově

orientovaných jazycích. Oficiální standard pro objektové databáze neexistuje. Jako standard se uvádí kniha od Morgana Kaufmana The Object Database Standard: ODMG-V2.0. Tyto databáze jsou stavěné pro práci s objektovými jazyky jako je Java, C++, C#, Python a Smalltalk. Představitelem této databáze je například db4o.

Konstruktorem v objektovém modelu je sám objekt, který se přímo ukládá do databáze, ulehčuje to především práci programátorů, kteří se nemusí učit nový jazyk.

Vlastnosti objektově orientované databází jsou podobné jako vlastnosti objektově orientovaných jazyků. Například musí plně podporovat objekty, třídy, zapouzdření, dědičnost, polymorfismus, reference mezi objekty a jednoznačnou identifikaci objektů.

Výhodou této databáze je objektový přístup, který umožňuje pohodlnou manipulaci s daty. Nevýhodou je pomalé a neefektivní zpracování dat z pohledu výkonu.

2.2.3 Objektově relační databáze

Objektově relační databáze (Object Relational Database Management System, ORDBMS) kombinují přístup předchozích dvou databází. V základu se jedná o relační databázi. Data se stále ukládají do tabulek, ale položky v tabulkách mohou mít bohatší datovou strukturu nazývanou abstraktní datový typ (ADT). Tento nový typ vznikne zkombinováním dosavadních datových typů. Podpora ADT je atraktivní. Operace a funkce spojené s novým datovým typem mohou být použity k indexování ukládaných dat a získávání záznamů na základě obsahu nového datového typu. Problémem je, že mají omezenou podporu dědičnosti, polymorfismu, referencí a integrace do programovacího jazyka. Toto omezení vyplývá z implementace. ORDBMS je podmnožinou RDBMS, pokud bychom nepoužili nový datový typ, pak je kompatibilní s relačním modelem. Specifikace je v rozšíření SQL standardu SQL3.

Konstruktory jsou stejné jako u relační databáze.

Výhodou ORDBMS je širší možnost využití a slučování dotazu. Nevýhodou je složitost, ze které vyplývá pomalejší a méně efektivní výkon dotazování.

2.2.4 Dokumentově orientované databáze

Dokumentově orientované databáze se ukládají do dokumentů na místo do tabulek jako je tomu u relačních databází a dotazování se provádí pomocí pohledů. Tyto pohledy jsou většinou v databázích ukládaný a jejích indexy plynule obnovovány. Představitelé jsou Apache CouchDB, MongoDB a Lotus Notes.

Konstruktory v těchto databázích bývají dokumenty a kolekce dokumentů vložená v dokumentu. Dokument může představovat například JOSN formát pro specifikaci JavaScript objektů.

Dokumentově orientované databáze má jednoduché použití v objektovém návrhu aplikace. Databáze nevyžaduje naučit se mnoho nových příkazů a práce s ní je jednoduchá. Tyto databáze se používají převážně u webových technologií. Taková databáze není vhodná pro ukládání obrovského množství dat s velkým počtem vztahů.

2.2.5 Grafové databáze

Grafové databáze jsou NoSQL (nerelační datová úložiště). Pojem ACID není pro grafové databáze závazný. Dbají především na výkon databáze, tedy efektivitu a rychlost dotazování, což je pro ně důležitější než konzistence.

Konstruktorem v grafové databázi je obecný graf, který má datová struktura nemající začátek ani konec. Je definována pomocí vrcholů a hran. Grafy dělíme na orientované a neorientované. Orientaci v grafu poznáme podle hran, které nám určují směr z jednoho vrcholu do druhého. Představiteli grafových databází jsou například Sones GraphDB, AllegroGraph, Neo4j, FlockDB, InfiniteGraph a Titan.

Obecný graf obsahuje neprázdnou množinu vrcholu disjunktní s množinou hran a neprázdnou množinu hran disjunktní s množinou vrcholů.

Grafová databáze dokáže uchovávat veliké množství dat a efektivně s nimi pracovat. Tyto vlastnosti jsme od databáze požadovali, proto si tento typ databáze vybereme. Konzistence dat není pro nás nejdůležitější vlastností. Ještě nám zbývá vybrat si vhodného představitele tohoto typu databáze.

2.3 Výběr grafové databáze

Hledáme grafovou databázi, která nám bude vyhovovat pro použití v aplikaci. Představíme si několik grafových databází a na konci vyhodnotíme, která nám nejvíce vyhovuje. Posuzovat budeme implementační jazyk, API databáze, licenci a její vlastnosti.

2.3.1 Sones GraphDB

Sones GraphDB je vyvíjen v jazyku C# německou společností Sones. Struktura této databáze se přirovnává k váženému grafu. Jinými slovy hrany jsou ohodnoceny reálnými čísly.

Databázi lze provozovat pod .NET frameworkem i pod Mono (zajišťuje multiplatformnost). Pro přístup k datům je možné použít jedno z mnoha rozhraní Java, C#, WebShell, WebDAV nebo REST API (Representational State Transfer Application Programming Interface), které nám zaručí nekomplikovaný přístup vzhledem ke zvoleným technologiím aplikace.

Licence pro Sones GraphDB je šířena ve dvou variantách. První je komunitní verze (open source AGPLv3). Druhá je komerční licence enterprise, která má širší možnosti použití.

2.3.2 AllegroGraph

AlleqroGraph je proprietární grafová databáze vytvořená vývojáři z Franz, Inc. Jde o moderní vysoce výkonnou, trvanlivou grafovou databázi. Ta efektivně využívá paměť využitím kombinací disků.

Databáze je navržena tak, aby splňovala standarty W3C (World Wide Web Consortium) pro Resource Description Framework, který je určen pro manipulaci s propojenými daty a sémantickým webem. Pro použití nabízí několik rozhraní v C#, Perl, Ruby a Scala.

AllegroGraph je šířena pod třemi licencemi a to Free, Develeper, Enterprise. První dvě licence jsou značne omezené a třetí je komerční, kterou využívají velké společnosti jako je Ford, Kodak a NASA.

2.3.3 Neo4j

Neo4j je asi nejpoužívanější z grafových databází. Je implementovaná v Javě s licencí open source. Pochází od společnosti Neo Technology, které má velkou vývojářskou komunita a dále ji rozvíjí.

Ukládání grafové struktury je v podobě hran, vrcholů a vlastností. Limit vlastností v grafu je nastaven až na několik desítek miliard. Každý vrchol nebo hrana může obsahovat

vlastnosti, které jsou uložené ve formátu klíč–hodnota. Tato grafová databáze se také nazývá *property graph database*. Neo4j se dá použít dvěma způsoby. Jedním způsobem je vestavěním do aplikace. Druhým způsobem je přístup přes REST API.

2.3.4 FlockDB

FlockDB je vyvíjena Twitterem pod licencí Apache License, Version 2.0, je určena pro analýzu souvisejících vztahů. Zatím nebyla vydaná stabilní verze.

Tato grafová databáze se vydala netradiční cestou, nebude v ní klasické traverzování, což je specifické pro grafové databáze. Někteří dokonce tvrdí že se nejedná o grafovou databázi, když traverzování neobsahuje. FlockDB obsahuje maximální podporu pro operace *add, update, remove*. Dále podporuje aritmetické operace nad množinou výsledků, jako je stránkování seznamu výsledků v seznamech (až o miliónech záznamů). Ve výsledku by měla podporovat horizontální škálování, replikací a migraci dat.

2.3.5 InfiniteGraph

InfiniteGraph je implementována v jazyce Java mimo jádro,které je v C++. Databázi vyvinula společnost Objectivity. Databázová struktura grafu je objektově orientovaná. Skládá se z objektů, které znázorňují vrcholy a hrany propojují vrcholy. Přístup k databázi je zatím dostupný pouze přes rozhraní Java. REST API zatím není k dispozici.

Licence pro InfiniteGraph je ve dvou variantách. Bezplatná, která je značně limitovaná a komerční, kterou používaná vláda spojených států amerických a poradenská společnost Deloitte.

2.3.6 Titan

Titan je uspořádaná grafová databáze optimalizovaná pro řazení a dotazování do grafů, obsahující až stovky bilionů vrcholů a hran. Multiplatformnost zajišťuje implementační jazyk Java. Titan je transakční databáze, která podporuje tisíce souběžně pracujících uživatelů s vyhodnocováním v reálném čase.

Titan je přizpůsoben pro uspořádání rostoucího počtu dat a uživatelů. Podporuje ACID. Nativně je integrován do frameworku TinkerPop[7] obsahujícího dotazovací jazyk Gremlin[8], framework Frames[2] (pro převod mezi objekty grafu a objekty Javy), framework Rexster[3] (grafový server) a Blueprints[4] (standardní grafové API).

Titán je distribuován jako open source s licencí Apache2.

2.3.7 Zhodnocení

Jako první jsme uvažovali o databázi Sones GraphDB. Tato databáze nám nevyhovuje z hlediska návrhu i po stránce licenční. Pro plnou podporu bychom museli platit.

Další databází je AllegroGraph. Tato databáze je na tom podobně, její návrh je určen pro webové technologie a pro plnou funkcionalitu bychom si jí museli koupit.

Neo4j odpovídá naším požadavkům jak z hlediska jazyka tak i po stránce výkonnostní. Proto jí zařadíme mezi hlavní kandidáty.

Databáze FlockBD si určuje vysoké nároky svých vlastností a její konstrukce neodpovídá naším požadavkům. Tuto databázi zavrhujeme protože nemá ještě stabilní verzi.

InfiniteGraph databáze je napsaná v Javě, místo vrcholu má objekty, důvodem proč jí nepoužijeme je, že za plnou licenci se musí platit.

Poslední databází, která nám zbývá je Titan, tato databáze má spoustu výhod. Umožňuje efektivní práci s velkým množstvím dat a možnost použití přímo v aplikaci. Dále podporuje nativně framework TinkerPop pro pohodlnější práci.

Z nabízených možností nám zbyli dvě. Neo4j a Titan, obě splňují naše požadavky. Nakonec jsme vybrali Titan. Hlavním důvodem našeho rozhodnutí je usnadnění práce díky frameworku TinkerPop.

2.4 Výběr dekompilátoru

V našem projektu budeme potřebovat dekompilátor pro získání byte kódu. Dekompilátorů Java `class` souborů není mnoho a většina je zastaralá nebo obsahují plno chyb. Zde si jích pár zhodnotíme a vybereme nejvhodnější.

CFR tento dekompilátor je v neustálém vývoji. V referenci na něj jsem se setkal spíše s negativními ohlasy.

Krakatau je napsaný v Pythonu. Je stále ve vývoji a jeho výsledky nejsou uspokojivé.

JD-GUI tento dekompilátor pracuje dobře, ale ještě nemá plnou podporu pro lamda výrazy, které byly do Javy nově přidány. Tento dekompilátor je taktéž v neustálém vývoji a je možné, že se toho hodně změnilo.

JAD tento dekompilátor pracuje dobře, pouze pro starší verze souborů Java 1.4 a nižší.

Procyon je z dosud vyvíjených dekompilátorů nejlepší. Je v aktivním vývoji a má už plnou podporu pro Java 8. Mohou se tam vyskytnout chyby a stabilní verze ještě nebyla uvolněna. Přesto je pro nás nejlepší volbou. Pro podrobnější popis Procyonu si vyhradím celou kapitolu, kde provedu rešerši tohoto nástroje.

2.5 Překladač jazyka

Pro náš jazyk budeme potřebovat překladač, jelikož tvorba překladače není snadná, použijeme nějaký generátor pro vygenerování překladače. Jako generátor analyzátoru mi byl doporučen ANTLR[6].

2.5.1 ANTLR

Another Tool for Language Recognition (jiný nástroj pro rozpoznání jazyka) jedná se o nástroj, který umožňuje generování vlastního překladače jazyka typu LL(*). ANTLR nám vygeneruje první dvě části překladače[5]: lexikální analyzátor a syntaktický analyzátor. Sémantickou kontrolu můžeme udělat třemi způsoby: zápisem přímo v syntaktickém analyzátoru (embeded), implementací posluchačů (listener), nebo pomocí návštěvníků (visitor).

Vykonání jazyku můžeme udělat se sémantickou analýzou nebo si můžeme vygenerovat příkazy a provést je až po sémantické analýze. Druhý způsob je vhodnější, protože oddělíme kontrolu jazyka od jeho interpretace. Pokud by nastala chyba v sémantické analýze, můžeme dotaz přerušit před jeho vykonáváním.

Kapitola 3

Rešerše Procyon

Procyon[9] je projekt zaměřující se na dekompilaci Java kódu. Vyvíjí ho Mike Strobel pod licencí The Apache Software License. Aktuální verze je 0.5.28. Název projektu je odvozen od souhvězdí, které nese stejný název Procyon.

3.1 Knihovny

Procyon obsahuje následující knihovny: `CoreFramework`, `Reflection Framework`, `Expressions Framework`, `CompilerToolset` a `JavaDecompiler`. Tyto knihovny jsou dostupné z Maven centra pod identifikátorem `org.bitbucket.mstrobel`. Nás bude zajímat pouze poslední knihovna `JavaDecompiler`.

3.2 Vlastnosti

Procyon dokáže analyzovat zdrojové kódy Javy verze 5 a vyšší. Procyon oproti ostatním dekompilátorům mnohem lépe zvládá dekompilaci některých částí Java kódu. Například dekompilaci deklarace výčtového typu, použití řetězce a výčtového typu v `switch` příkazu, umístění tříd jak anonymních tak pojmenovaných, anotací, ukazatele na lambda příkazy[1].

3.3 Konzole

Procyon můžeme spustit, jako konzolovou aplikaci ve třech režimech. Ve výchozím nastavením získá zdrojové kódy Java. Druhý režimem získáme neuspořádaný byte kód při použití argumentu `-r`. Výstup je podobný jako při použití nástroje `javap`, ale hezčí jak uvádí autor. Ve třetím režimu získáme byte kód uspořádaný do AST (abstraktního syntaktického stromu) a to použitím argumentu `-b` a pro optimalizovaný AST použijeme argument `-u`.

3.4 API

Pro použití Procyonu ve své aplikaci použijeme API Procyonu. Hlavní třídou pro API je `com.strobel.decompiler.DecompilerDriver`. Lze také použít `com.strobel.decompiler`

[1] Srovnání výstupů dekompilátoru si můžete prohlédnout na: `https://bitbucket.org/mstrobel/procyon/wiki/Decompiler%20Output%20Comparison`

`.Decompiler` a výstup jde momentálně do souboru nebo konzole. Pokud chcete výstup do proměnné, musíte ho přesměrovat ručně[2].

3.5 GUI

Pokud nechcete používat příkazovou řádku, můžete také použít grafické rozhraní. Máte na výběr ze dvou možností `Luyten` a `Bytecode Viewer`.

3.6 Alternativy

Mezi alternativy patří CFR od Leeho Benfielda, Karkatau od Roberta Grosse, Candle od Brada Davise a Fernflower od Roman Shevchenko. Ovšem tyto alternativy nejsou tak dokonalé jako Procyon.

[2]Více o použití API najdete na: `https://bitbucket.org/mstrobel/procyon/wiki/Decompiler%20API`

Kapitola 4

Návrh aplikace

V druhé kapitole jsme analyzovali prostředky, jak dosáhneme výsledku. V této kapitole si navrhneme dotazovací jazyk. Následně navrhneme strukturu grafové databáze a nakonec navrhneme strukturu aplikace pomocí diagramu balíčků a sekvenčního diagramu.

4.1 Návrh jazyka

Návrh jazyka uvedeme v gramatické specifikaci pro ANTLR verzi 4. Při návrhu jsme se inspirovali klíčovými slovy z SQL syntaxe. Nebude to zrovna typická struktura jazyka, použijeme-li grafovou databázi a ne tabulkovou. Důvodem proč jsme se takto rozhodlo, je rozšířenost jazyka SQL mezi programátory a jeho snadná pochopitelnost. Dále už budu popisovat gramatiku navrženého jazyka. Na pravé straně bude vždy neterminál a na levé náhrada za něj.

```
program : selectStatement ;
```

Pravidlo `program` je počáteční, které se použije na vstupní řetězec při stavbě derivačního stromu jako první. Na levé straně má pouze jedno pravidlo `selectStatement` (viz str. 10), které se nahradí za `program`. Pravidlo `selectStatement` je popsáno níže.

```
selectStatement :
        (SELECT paramSelect)?
                (FROM packages)?
                        (WHERE conditions)?
                                (UNIQUE)?;
```

Pravidlo `selectStatement` popisuje jednotlivé části dotazu. Na levé straně máme čtyři nepovinné klauzule, kde každý specifikuje jednu část vyhledávání. První z nich je `SELECT`[1] `paramSelect` (viz str. 11), kde `paramSelect` je pravidlo popisující co budeme požadovat z výsledných tříd. Druhý je `FROM packages` (viz str. 11), kde pravidlo `packages` popisuje, kde se bude vyhledávat v `jar` souboru. Třetí je `WHERE conditions` (viz str. 12), kde pravidlo

[1] Terminály začínají velkými písmeny a jsou definované v příloze na CD v souboru `query.g`.

`conditions` popisuje podmínky pro třídy. Čtvrté klíčové slovo `UNIQUE` určí, zda se mají vyhledávat pouze unikátní výsledky.

```
paramSelect : paramName (COMMA paramName)* ;
```

Pravidlo `paramSelect` popisuje, které záznamy budeme požadovat z tříd. Na levé straně je pravidlo typu `paramName` (viz str. 11), které může být zadané vícekrát navzájem oddělené čárkami.

```
paramName : EXCLAMANTION alias? NAME
          | alias? NAME (LBRACKET method RBRACKET)?
          | innerSelect
          | alias? STAR;
```

Pravidlo `paramName` má na výběr ze čtyř levých stran oddělených svislou čarou. První na výběr máme `EXCLAMANTION alias?` (viz str. 12) `NAME` tato levá strana se používá pro vztahy mezi třídami `EXCLAMANTION` říká, že chceme jít v opačném směru po hraně `NAME`. Druhá levá strana `alias? NAME (LBRACKET method` (viz str. 13) `RBRACKET)?` popisuje volání metod ve třídě, třetí variantou je `innerSelect` (viz str. 13), což je vnořený dotaz a čtvrtá varianta je `alias? STAR`, která vybere všechny aktuální třídy.

```
packages : packageLink (COMMA packageLink)*;
```

Pravidlo `packages` popisuje výběr tříd, se kterými se bude pracovat. Na levé straně je pravidlo `packageLink` (viz str. 11) specifikující výběr tříd navzájem oddělených čárkami.

```
packageLink : packageName (JOIN packageName)* as?
            | STAR as?;
```

Pravidlo `packageLink` spojí vybrané třídy z pravidla `packageName` (viz str. 11), kterým může přiřadit náhradní jména tzv. aliasy, a to pravidlem `as?` (viz str. 12). Druhá levá strana `STAR as?` vybere všechny třídy.

```
packageName : EXCLAMANTION? STRING
            | innerSelect ;
```

Pravidlo `packageName` popisuje výběr tříd. Buď jako cestu mezi balíky `EXCLAMANTION? STRING`, kterou určuje `STRING` a při uvedení terminálu `EXCALAMANTION` nevyhledáváme v podbalících. Nebo použijeme vnitřní dotaz `innerSelect` k výběru tříd.

```
as: AS NAME ;
```

Pravidlo `as` slouží k vytvoření zástupného jména pro množinu tříd tzv. alias.

```
alias : NAME_DOT ;
```

Pravidlo `alias` slouží k zpřístupnění množiny tříd přes zástupné jméno tzv. alias.

```
conditions : cond (AND cond)* ;
```

Pravidlo `conditions` popisuje spojení podmínek. Podmínky jsou spojeny logickým operátorem `AND`, který si podmiňuje splnění všech podmínek, aby mohl být výsledek pravdivý. V našem případě třída musí splňovat všechny podmínky.

```
cond : equal
     | (EXIST|NOT_EXIST) innerSelect
     | innerSelect OPERATORS alias? NAME
     | EXCLAMANTION? alias? NAME;
```

Pravidlo `cond` popisuje podmínky. První z možností podmínek je `equal` (viz str. 12). Další podmínkou je `(EXIST|NOT_EXIST) innerSelect`, která nás informuje, zda má vnitřní dotaz nějaké výsledky. Dalším druhem podmínky je `innerSelect OPERATORS alias? NAME` požadujeme, aby vnitřní dotaz měl souvislost s množinou záznamů `NAME`. Poslední podmínkou je `EXCLAMANTION? alias? NAME`. Jedná se o pravdivostní tvrzení, zda má třída určitou vlastnost nebo nemá, pokud je uvedeno `EXCALAMANTION`.

```
equal : alias? NAME OPERATORS rightStatement
      | alias? annotated OPERATORS rightStatement
      | alias? annotated
      | alias? METHOD LBRACKET method RBRACKET;
```

Pravidlo `equal` porovnává vlastnosti tříd. První variantou je `alias? NAME OPERATORS rightStatement` (viz str. 13). Zde porovnáváme vlastnosti třídy `NAME` a `rightStatement` způsob porovnání určíme operátorem `OPERATORS`. Druhou variantou je `alias? annotated` (viz str. 13) `OPERATORS rightStatement` hledáme užití anotace s parametrem ve třídě a tento parametr budeme porovnávat s `rigthStatement` způsobem podle operátoru `OPERATORS`. Třetí variantou `equal` je `alias? annotated` hledaní užití anotace ve třídě. Poslední variantou je `alias? METHOD LBRACKET method RBRACKET` hledání metody.

```
rightStatement : alias? NAME
```

```
        | alias? annotated
        | STRING
        | PATTERN;
```

Pravidlo `rightStatement` popisuje pravou stranu porovnání, na pravé straně se může vyskytovat `alias? NAME` vlastnost třídy, `alias? annotated` hodnota parametru anotace, `STRING` libovolný řetězec a `PATTERN` regulární výraz.

```
innerSelect : LPAREN selectStatement RPAREN ;
```

Pravidlo `innerSelect` umožňuje vnořování dotazů dotazy do libovolné hloubky.

```
annotated : (METHOD LBRACKET method RBRACKET index?)?
        annotatedStatement ;
```

Pravidlo `annotated` popisuje tvar užití anotace, pokud se anotace váže k metodě pak je před `annotatedStatement` (viz str. 13) uvedeno `(METHOD LBRACKET method RBRACKET index?` (viz str. 14)). Pokud vynecháme `indexu`, hledáme anotovanou metodu. V opačném případě hledáme anotovaný parametr metody. Jestliže hledáme anotovanou třídu, tak před `annotatedStatement` nic nepřidáváme.

```
annotatedStatement : AT_NAME (annotatedParams)* ;
```

Pravidlo `annotatedStatement` popisuje anotaci a její parametry. Terminál `AT_NAME` je název anotace a pravidlo `annotatedParams` specifikuje parametr anotace.

```
annotatedParams : DOT AT_NAME
        | DOT_NAME index? ;
```

Pravidlo `annotatedParams` popisuje parametr anotace. Parametrem anotace může být další anotace, název parametru nebo pole parametrů. Pokud je hledaný parametr zanořený, jednotlivé parametry jsou odděleny tečkou a u polí je přidané pravidlo `index`.

```
method : STRING
        | NAME OPERATORS STRING (COMMA NAME OPERATORS STRING)*
        | STAR;
```

Pravidlo `method` popisuje metodu, kterou hledáme. Pravidlo má tři varianty. První je `STRING` popis metody, druhý je `NAME OPERATORS STRING (COMMA NAME OPERATORS STRING)*`, kde `NAME` je vlastnost metody a `STRING` je její hodnota. Posledním je `STAR`, ta nehledá určitou metodu, ale bere všechny metody.

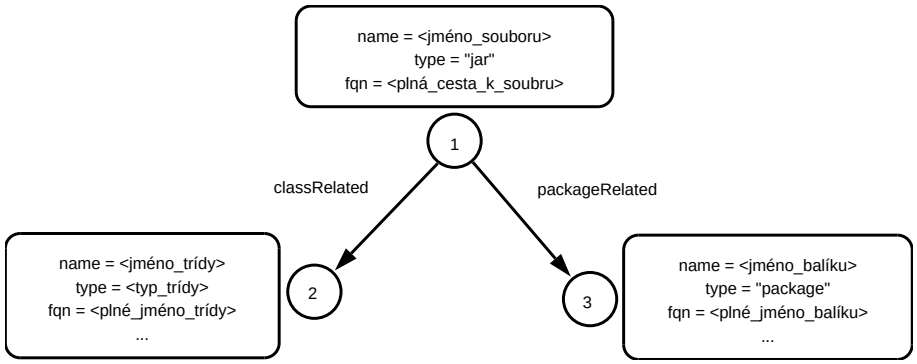
```
index : LBRACKET INT RBRACKET
      | LBRACKET STAR RBRACKET;
```

Pravidlo `index` popisuje indexaci položky. Varianty jsou dvě. Buď je `INT` kladná celočíselná hodnota, která určí položku. Nebo `STAR` bere v úvahu všechny položky.

Úplná gramatika je přiložená na CD v souboru `query.g`, kde jsou popsány tvary terminálů. Sémantika gramatiky bude popsána v kapitole o implementaci s ukázkami kódu sémantických akcí.

4.2 Návrh grafové databáze

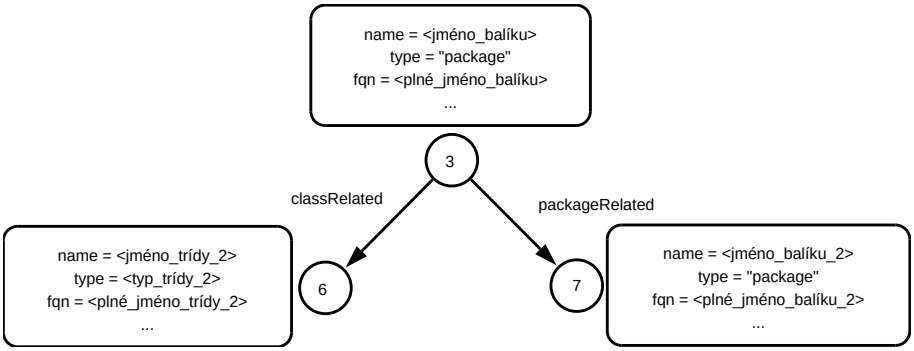
Návrh struktury grafové databáze je jednou z nejdůležitějších části této práce. Do grafu musíme být schopní uložit potřebné informace tak, abychom se na ně mohli rychle a efektivně dotazovat. Následující část popíše, jak jsme postupovali při návrhu struktury grafu. Začneme postupně popisovat vlastnosti vrcholů grafu a jejích propojení s dalšími vrcholy. Propojení se provádí pomocí orientovaných hran, které určují, o jaký vztah se jedná a kolik takových vztahů může daný vrchol nabývat. Vrcholy jsou znázorněny kružnicemi, které mají uprostřed číslici určující index vrcholu. U vrcholů jsou obdélníky s vlastnostmi vrcholu, pro přehlednost jsou uvedeny pouze tři. Ostatní vlastnosti jsou popsány v odstavci pod grafem. Jednotlivé vrcholy jsou propojeny šipkami s názvem vztahu.



Obrázek 4.1: Vrchol pro jar souboru.

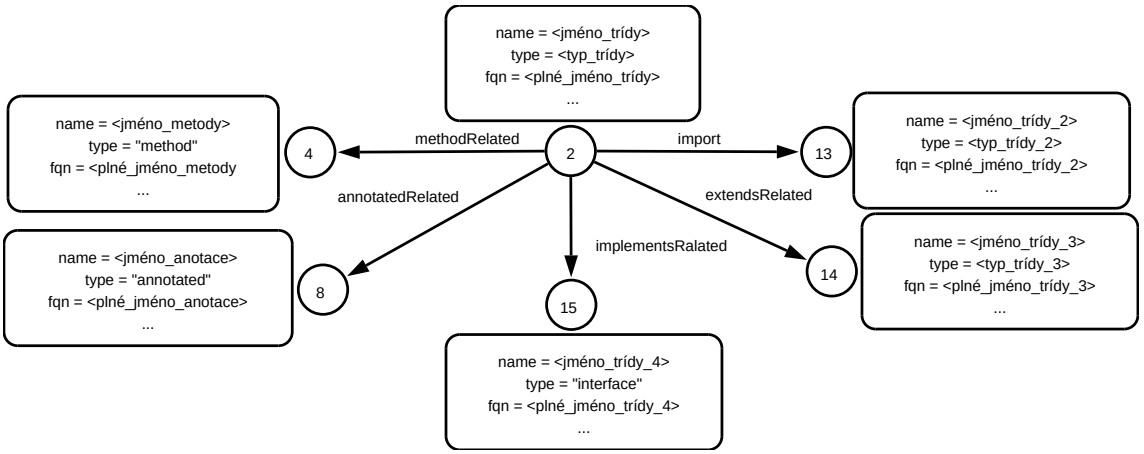
Prvním vrcholem a kořenem grafu bude vrchol s indexem `1` (viz obr. 4.1), který bude znázorňovat jar soubor. Informace, které budeme potřebovat uložit do tohoto vrcholu jsou název jar souboru, cesta k jar souboru a typ, který nám určí, o jaký vrchol jde. Tento typ souboru se bude v grafové databázi nacházet pouze jednou. K vrcholu s indexem `1` se vážou dva typy vrcholů pro třídy a balíky. Třídy jsou vázány pomocí hran `classRealted` směrem od vrcholu s indexem `1` k vrcholu s indexem `2`. Těchto hran může vrchol jar nabývat 0...N. Balíky jsou vázány pomocí hran `packageRelated` směrem od vrcholu s indexem `1` k vrcholu s indexem `3`. Těchto vztahů může vrchol s indexem `1` nabývat rovněž 0...N.

Vrchol pro balíky s indexem `3` (viz obr. 4.2) se napojuje na vrchol jar souboru nebo na vrchol jiného balíku. Informace které uchovává vrchol balíku, je typ vrcholu (u balíku je to vždy `package`), název balíku a název s cestou balíku. K vrcholu balíku se vážou dva typy vrcholů a to vrcholy třídy a vrchol podbalíku. Třída se napojuje na balík pomocí hrany



Obrázek 4.2: Vrchol pro balíky.

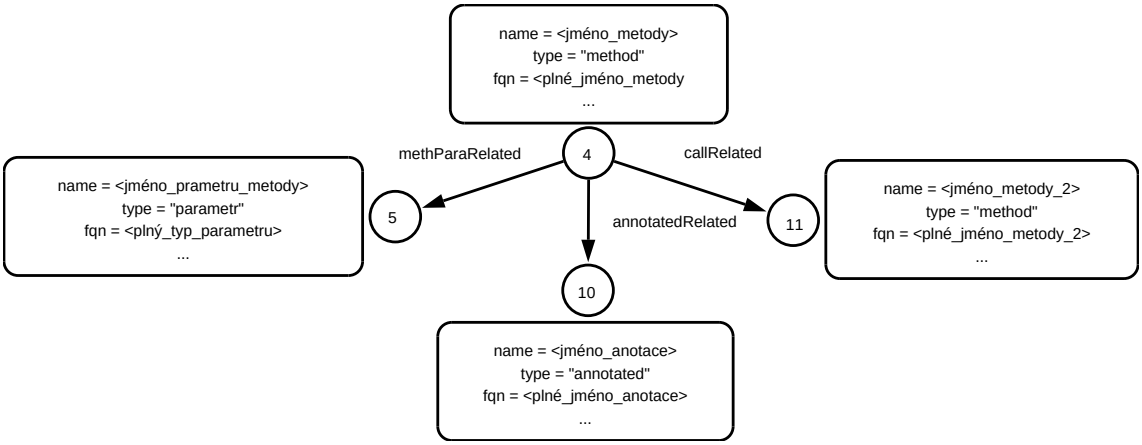
`classRelated`, těchto hran může mít 0...N. Podbalíky se vážou hranami `packageRelated`, kterých může mít 0...N.



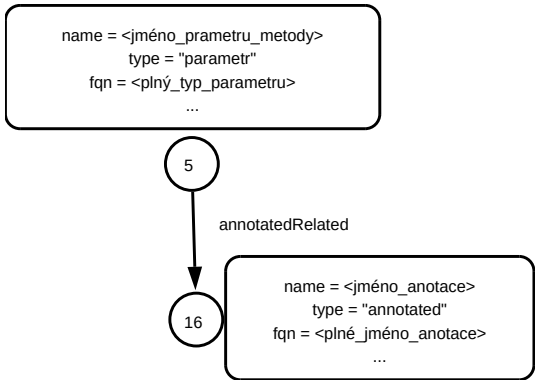
Obrázek 4.3: Vrchol pro třídy.

Vrchol třídy s indexem `2` (viz obr. 4.3) se napojuje na jar vrchol nebo vrchol balíku. Informace, které budeme ukládat do tohoto vrcholu, jsou název třídy, plný název třídy, typ třídy, popis třídy a modifikátory. K vrcholu třídy se vážou tři typy vrcholů. Prvním vrcholem může být jiný vrchol třídy, který se váže pomocí jednoho ze tří typů hran. První hranou je `extendsRelated`, tento vztah může mít třída pouze jeden. Druhou hranou je `implementsRelated`, těchto vztahů může být více a třetí je `importRelated`, těchto vztahů může být rovněž více. Druhým typem vrcholů, který se váže k vrcholu třídy, jsou vrcholy metod. Metody jsou vázány pomocí hran `methodRelated`. Těchto vztahů může vrchol třídy nabývat 0...N. Posledním vrcholem, který se váže k vrcholům tříd jsou vrcholy anotace a to hranou `annotatedRelated`, těchto hran může mít vrchol třídy 0...N.

Vrchol pro metodu s indexem `4` (viz obr. 4.4) se napojuje na vrcholy tříd. Informace, které uchovává jsou: název metody, typ (v tomto případě bude vždy `method`), popis metody, stručný popis metody, počet parametrů a modifikátory. K vrcholu metody se vážou tři typy vrcholů. První je parametr metody vázané hranou `methParaRelated`, každý vrchol metody může mít libovolný počet těchto vztahů. Druhý vrchol je anotace vázaná hranou `annotatedRelated`, těchto hran muže mít rovněž libovolný počet. Posledním vrcholem je vrchol metody vázaný hranou `callRelated`, počet těchto hran je libovolný.



Obrázek 4.4: Vrchol pro metody.



Obrázek 4.5: Vrchol pro parametry metody.

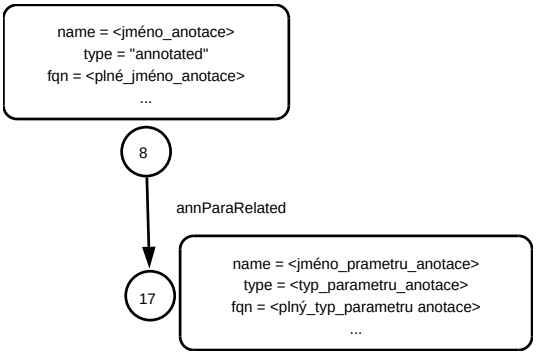
Vrchol pro parametry metody s indexem `5` (viz obr. 4.5) se napojuje na vrchol metody. Informace, které uchovává jsou: název parametru, typ vrcholu (v tomto případě `parametr`), index (kolikátý je to parametr) a úplný typ parametru. K vrcholu parametru se váže jeden typ vrcholu a to vrchol anotace. Tato anotace se váže hranou `annotatedRelated`, těchto hran může vrchol parametru metody nabývat 0...N.

Vrchol pro anotace s indexem `5` (viz obr. 4.6) se napojuje na vrcholy tříd a metod. Informace, které uchovává jsou: název anotace, typ vrcholu (v tomto případě `annotated`), plný název anotace a popis anotace. K vrcholu anotace se váže jeden typ vrcholu a to vrchol parametru anotace. Tento vrchol je napojen hranou `annParaRelated`, těchto hran může mít 0...N.

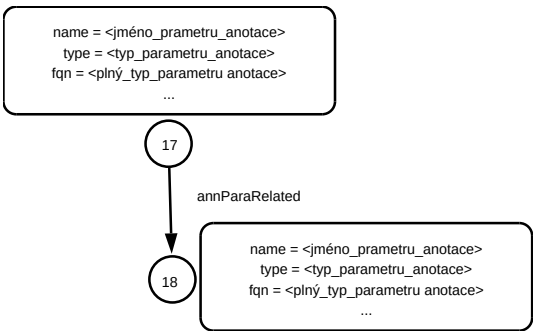
Posledním vrcholem, který se může vyskytovat v grafu je parametr anotace s indexem `17` (viz obr. 4.7). Informace, které uchovává jsou: název parametru, hodnota parametru a typ parametru. K vrcholu parametru anotace se váže jeden typ vrcholu a to vrchol parametru anotace. Tento vrchol se napojuje hranou `annParaRelated`, těchto hran může mít 0...N.

4.3 Návrh struktury aplikace

V této sekci se budeme zabývat návrhem struktury aplikace. Nejprve si program rozčleníme na logické části, které budou vykonávat určité úkony. Tyto části si popíšeme a vytvoříme pro



Obrázek 4.6: Vrchol pro anotace.



Obrázek 4.7: Vrchol pro parametr anotace.

ně balíčky. V dalším části budeme popisovat sekvenční diagram, kde si ukážeme komunikaci mezi třídami při spuštění aplikace.

4.3.1 Diagram balíčků

Celá aplikace bude zapouzdřena v balíčku `com.queryToAST.app` (viz obr. 4.8). Vnitřek balíčku je rozdělen do několika větších a menších balíčků. Každý balíček zapouzdřuje nějakou logickou funkcionalitu. Funkcionalita, kterou budeme zapouzdřovat obsahuje: spuštění aplikace, pomocné třídy obecného použití, získání abstraktního syntaktického stromu, získání metadat, práce s grafem a interpret jazyka.

Ke spuštění aplikace slouží balíček `API` (viz obr. 4.8). V tomto balíčku se budou nacházet dvě třídy. První bude sloužit pro spuštění aplikace. Druhá třída bude sloužit pro nastavení a řízení aplikace. A pomocné třídy budou umístěny v balíčku `Core` (viz obr. 4.8).

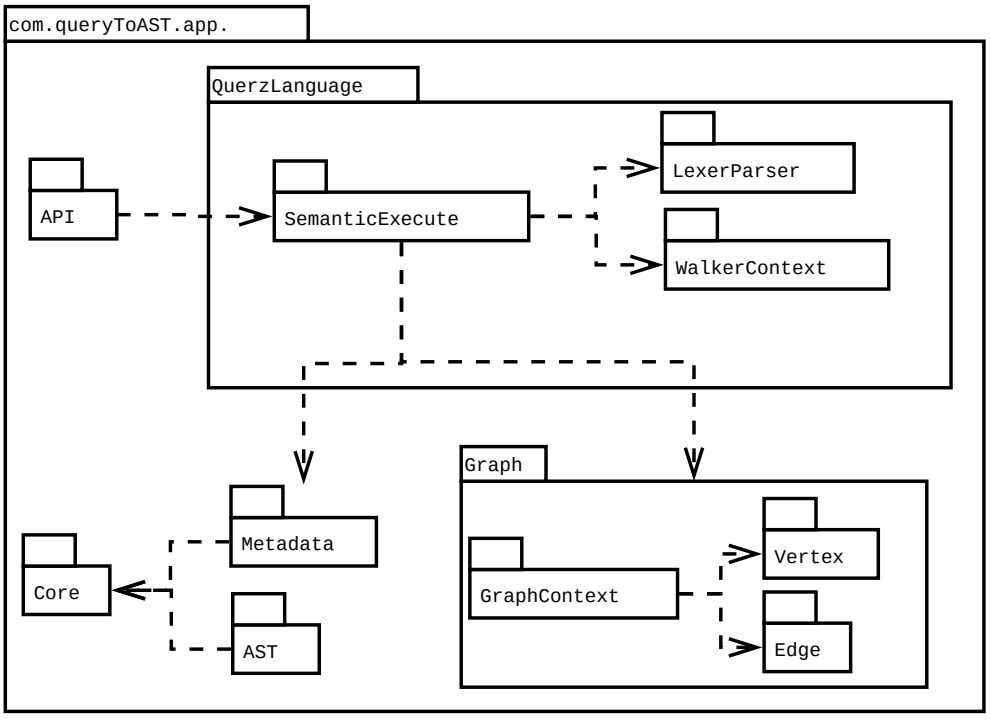
Získání abstraktního syntaktického stromu bude v balíčku `AST`. Tento balíček uvádíme pouze pro úplnost[2].

Získání metadat je umístěno v balíčku `Metadata` (viz obr. 4.8). V tomto balíčku je uveden postup dekompilace a následné zpracování výsledků.

Práce s grafem bude zapouzdřena v balíčku `Graph` (viz obr. 4.8). Zde bude tvorba a manipulace s grafem a vytvoření objektů pro hrany a vrcholy.

Poslední částí, kterou chceme zapouzdřit, je interpret jazyka. Ten bude umístěn v balíčku `QueryLanguage` (viz obr. 4.8). V tomto balíčku se budou nacházet třídy provádějící fáze překladu jako jsou: lexikální analýza, syntaktická analýza, sémantická analýza, generátor

[2]V aplikaci nemusíme tento balíček použít.



Obrázek 4.8: Diagram balíčku.

vnitřního kódu a interpret.

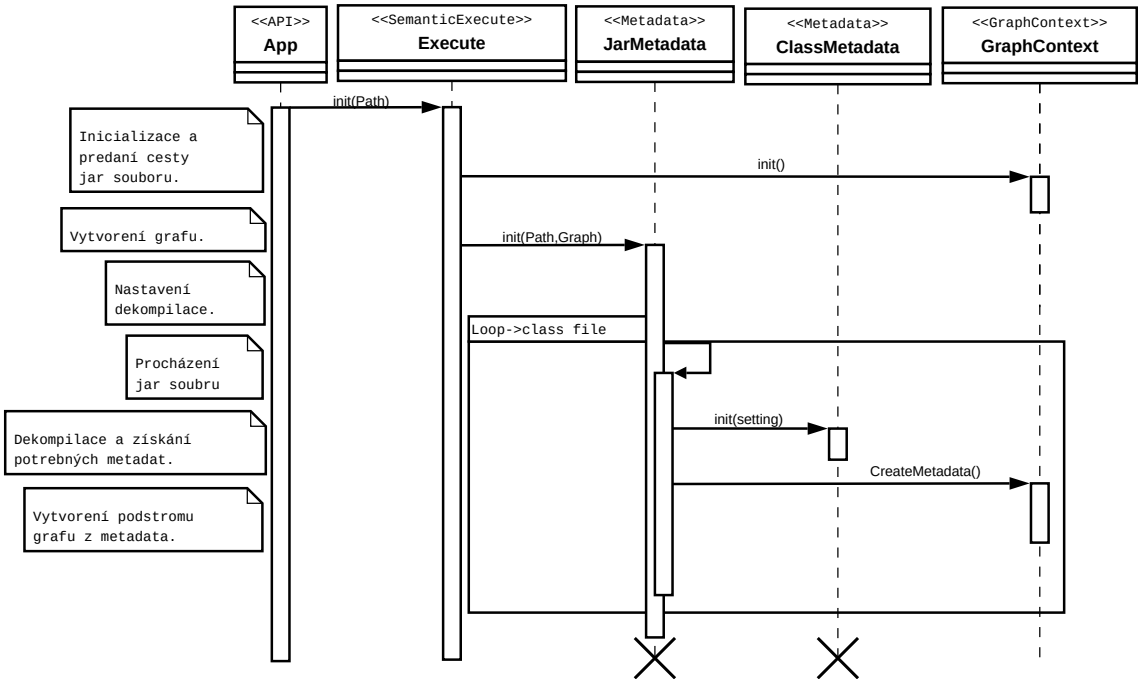
4.3.2 Sekvenční diagram

Když jsme si popsali, jak bude aplikace rozdělena, musíme popsat komunikaci v aplikaci. Komunikaci popíšeme v sekvenčním diagramu. Začneme komunikací při tvorbě grafu (viz obr. 4.9) a poté navážeme komunikací při položení dotazu (viz obr. 4.10). V sekvenčních diagramech budou mezi sebou komunikovat třídy, které budou mít ve stereotypu uvedené umístění v jar souboru. V sekvenčním diagramu jsou uvedeny pouze důležité třídy, na kterých závisí běh aplikace.

Sekvenční diagram tvorby grafu (viz obr. 4.9) popisuje proces od spuštění aplikace až po vytvoření grafové databáze. Nejprve `App` inicializuje `Execute`, tím že mu pošle zprávu s cestou k jar souboru.

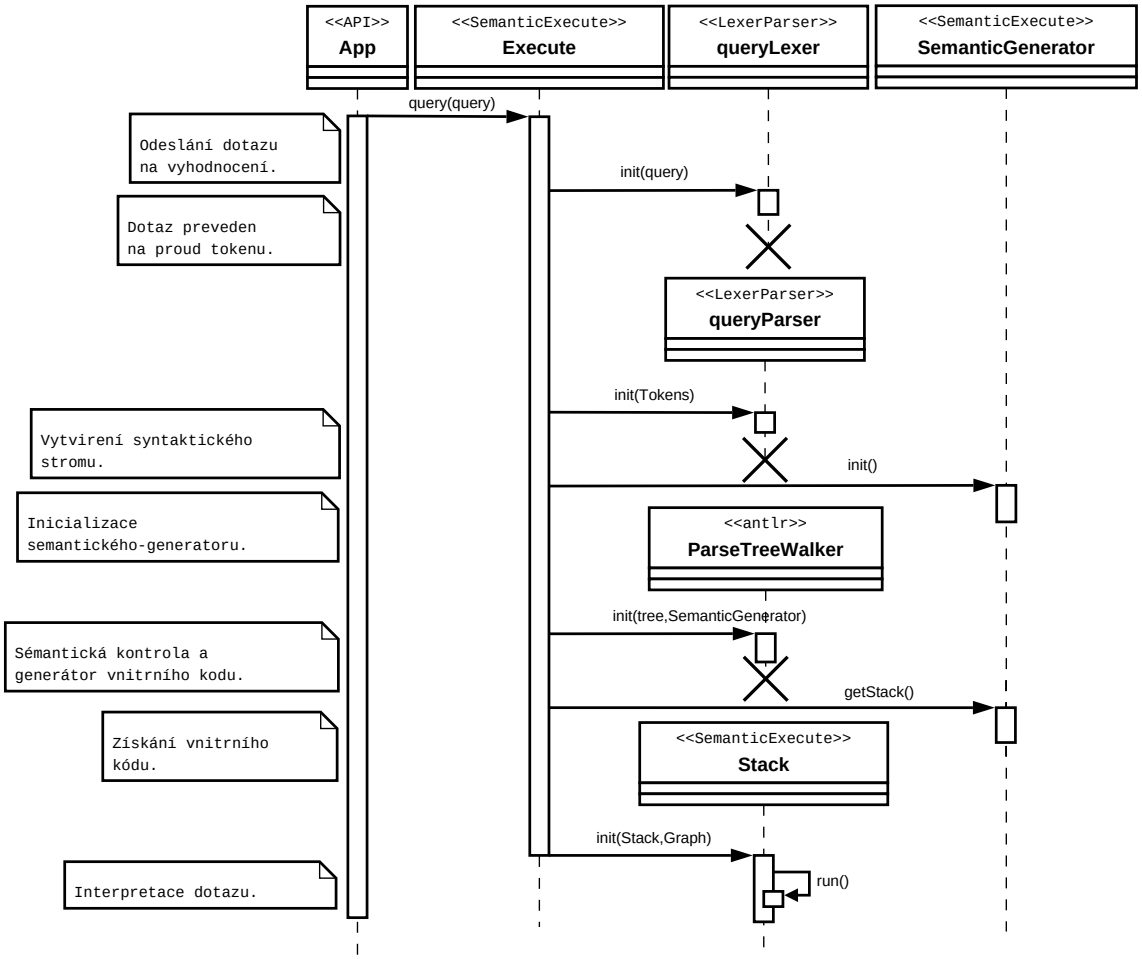
Poté vytvoříme prázdnou grafovou databázi pomocí třídy `GraphContext` a uložíme její kontext. Následně inicializujeme třídu `JarMetadata` s parametrem cesty k jar souboru a kontextem na grafovou databázi. Při inicializaci třídy `JarMetadata` zavolá vlastní metodu, která nastaví postup dekompilace. A v cyklu projde class soubory z jar souboru. V každém cyklu dekompilujeme aktuální class soubor pomocí třídy `ClassMetadata`. Dekompilovaná data uložíme do databáze. Po skončení cyklu máme naplněnou databázi, do které můžeme začít pokládat dotazy.

Vyhodnocení dotazu probíhá následovně (viz obr. 4.10). Prvním krokem je odeslání dotazu třídě `Execute`. Třída `Execute` provede lexikální analýzu třídou `queryLexer` a syntaktickou analýzu třídou `queryParser`. Z třídy `queryParser` získáme syntaktický strom, který použijeme při sémantické analýze. Sémantické akce jsou popsané ve třídě `SémanticGenerátor`, která také generuje instrukce pro vykonání dotazu. K vykonání sé-



Obrázek 4.9: Sekvenční diagram tvorby grafu.

mantických akcí použijeme třídu `ParserTreeWalker`, která projde syntaktický strom a zavolá příslušné sémantické akce. Po vykonání sémantických akcí požádáme třídu `SémanticGenerátor`, aby nám vrátila seznam instrukcí uložený v třídě `Stack`. Nad třídou `Stack` poté zavoláme metodu pro vykonání instrukcí, čímž dosáhneme vyhodnocení dotazu.



Obrázek 4.10: Sekvenční diagram položení dotazu.

Kapitola 5

Implementace aplikace

V předchozí kapitole jsme navrhovali gramatiku jazyka a strukturu aplikace. V této kapitole budeme popisovat konkrétní implementaci. Začneme základní prací s aplikací. Poté popíšeme postup při dekompilaci a tvorbě grafové databáze. Nakonec popíšeme implementaci interpretu jazyka.

5.1 Základní práce s aplikací

Aplikace je koncipována jako knihovna. Pro rychlé použití jí můžete spustit jako konzolovou aplikaci. Hlavní třídou pro spuštění je `com.queryToAST.app.API.App`. Třída `App` přijímá dva typy argumentů. Jsou to `-help` pro nápovědu aplikace a `-console` pro spuštění aplikace.

V nápovědě aplikace jsou informace o projektu (autor, název, rok atd.), popis argumentů pro spuštění aplikace a návod pro tvorbu dotazů.

Při spuštění budeme vyzváni k zadání cesty k jar souboru, ve kterém chceme vyhledávat třídy. Po potvrzení cesty jar souboru, bude provedena dekompilace a vytvoří se grafová databáze. Poté můžete zadávat dotazy pro vyhledání tříd. Výsledkem bude výpis plně kvalifikovaných názvů tříd ve formátu `FQN:com.test.Test`. Pokud nastane chyba ve struktuře dotazu, budou vypsána chybová hlášení. Dotaz můžeme opravit a znovu položit. Ilustrační příklad spuštěné aplikace můžete vidět níže.

```
Zadejte cestu k jar souboru: C:\project\Test.jar
<:SELECT * FROM * WHERE name='Vehicle'
FQN:com.test.Vehicle
FQN:com.test2.Vehicle
<:
```

Na příkladu můžete vidět zadání cesty k jar souboru `Test.jar` ve složce `C:\project`. Následně položení dotazu `SELECT * FROM * WHERE name='Vehicle'` a jeho výsledek `FQN:com.test.Vehicle, FQN:com.test2.Vehicle`. Nakonec ukončíme aplikaci zadáním prázdného dotazu.

```
public void console(){
        System.out.print("Zadejte cestu k jar souboru:");
        Scanner in = new Scanner(System.in);
```

```
        String internalName = in.nextLine();
        execute exec = new execute(internalName);
        while(true) {
                System.out.print("<: ");
                String query = in.nextLine();
                if(query.compareTo("") == 0) {
                        break;
                }
                List<ClassEntity> result = exec.query(query);
                for(ClassEntity ce : result) {
                        System.out.println("FQN : " + ce.getFQN());
                }
        }
}
```

Tento příklad uvádí implementaci konzolové aplikace. Do proměnné `internalName` načteme cestu jar souboru z konzole. Proměnou `internalName` použijeme jako parametr při inicializaci třídy `execute`. Třída `execute` při inicializaci dekompiluje jar soubor a vytvoří databázi. Následně čteme ve smyčce z konzole vstup, který uložíme do proměnné `query`. Pokud je proměnná `query` prázdná, ukončíme aplikaci. V opačném případě zavoláme metodu `query(query)` z třídy `execute`, která vyhodnotí dotaz. Výsledkem metody `query(query)` je seznam výsledných vrcholu tříd, který projdeme v cyklu a vypíšeme do konzole.

Jak je vidět, tak použití knihovny je velice jednoduché. V konzolové aplikaci jsme použili pouze třídu `execute` a seznam pro uložení tříd.

5.2 Implementace API knihovny

V předchozí části jsme si ukázali základní práci s knihovnou k vytvoření konzolové aplikace. Teď si popíšeme implementaci této třídy `execute`, abychom pochopili její vnitřní strukturu.

5.2.1 Příprava před položením dotazu

Před položením dotazu, musíme inicializovat databázi a naplnit jí.

```
public execute(String path) throws IOException {
        _graphContext = new GraphContext();
        new JarMetadata(path, _graphContext);
}
```

Konstruktor třídy `execute` přijímá řetězec s cestou k jar souboru. Prvním krokem konstruktoru je vytvoření prázdné grafové databáze (viz sekce 5.4) a uložení kontextu na databázi. Dalším krokem je naplnění databáze. Databázi naplníme pomocí třídy `JarMetadata`(viz sekce 5.3), které předáme cestu k jar souboru a kontext na databázi.

5.2.2 Zpracování dotazu

Zpracování dotazu provádí metoda `query(String query)` z třídy `execute`. Tato metoda provede nezbytné kroky k překladu dotazu a jeho vyhodnocení. Těmito kroky jsou: lexikální analýza, syntaktická analýza, sémantická analýza, generátor instrukcí a interpret jazyka.

```
public List<ClassEntity> query(String query){
    ANTLRInputStream input = new ANTLRInputStream(query);
    queryLexer lexer = new queryLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    queryParser parser = new queryParser(tokens);
    ParseTree tree = parser.program();
    ParseTreeWalker walker = new ParseTreeWalker();
    SemanticGenerator semGen = new SemanticGenerator();
    walker.walk(semGen, tree);
    semGen.PrintErr();
        Stack stack = semGen.getStack();
        stack.setGraphContext(_graphContext);
        return stack.run();
}
```

Při vyhodnocení dotazu získáme dotaz ve formátu řetězce. Řetězec převedeme na proud znaků pomocí třídy `ANTLRInputStearm`, který pošleme třídě `queryLexer` (lexikální jednotka). Třída `queryLexer` nám vrátí jednotlivé lexémy, které převedeme na proud tokenů třídou `CommonTokenStream`. Tento proud tokenů pošleme třídě queryParser a zavoláme metodu program. Metoda program nám vrátí syntaktický strom. Tento vygenerovaný syntaktický strom nadále budeme chtít procházet pomocí třídy `ParserTreeWalker`. Pro průchod stromem potřebujeme posluchače (listenery) a syntaktický strom. Posluchač neboli také listener je jeden ze způsobů jak provádět operace nad stromem. Třída `SemanticGenerator` je posluchač, který provádí sémantické akce a generování instrukcí. Po vytvoření instance posluchače (SemanticGenerator) můžeme zahájit průchod stromem a to metodou `walk(semGen,tree)`, která je součástí třídy `ParserTreeWalker`. Po ukončení průchodu vytiskneme chybová hlášení, pokud nějaká nastala. Následně získáme seznam vnitřních instrukcí a uložíme je do třídy `Stack`. Třídě `Stack` nastavíme kontext na databázi pomocí metody
`setGraphContext(_graphContext)`. Po nastavení kontextu na graf můžeme spustit interpret dotazu metodou `run()`. Výsledkem metody `run()` je seznam vrcholů, který je výsledkem celé metody `query(String query)`.

5.3 Dekompilace

Pro naplnění databáze použijeme třídu `JarMetadata`. Tato třída se stará o získání metadat a byte kódu. K získání těchto dat použije dekompilátor. Tato data poté pošleme do třídy `GraphContext`, která je uloží do databáze.

5.3.1 Rozbalení jar souboru

Rozbalení jar souboru provádí třída `JarMetadata`. Jde především o získání class souborů z jar souboru a nastavení jejích dekompilace. Výsledná data ukládá do databáze. Důležité prvky implementace jsou popsány níže.

```
public JarMetadata(
        String _internalName, GraphContext graphContext) {
        _graphContext = graphContext;
    _settings = new Setting(_internalName,null);
    _settings.setMetadata(true);
    execute();
}
```

Konstruktor třídy `JarMetadata` požaduje parametry `internalName`(interní cesta k souboru) a kontext na grafovou databázi. Kontext na grafovou databázi uloží do privátní proměnné pro pozdější použití. Dále nastaví privátní proměnnou `_setting`, která řídí průběh dekompilace. Při volání konstruktoru `Setting` požadujeme jako parametry `_internálName` a cestu pro uložení dekompilovaného souboru. V našem případě nechceme dekompilovaný soubor ukládat a proto volíme hodnotu `null`. Po inicializaci nastavíme požadovaný druh dekompilace. V našem případě požadujeme získání metadat, což provedeme zavoláním metody `setMetadata(true)`. Nakonec zavoláme metodu `execute()` pro zahájení dekompilace jar souboru.

```
private void execute() throws IOException {
        DecompilerSettings settings;
        settings = DecompilerSettings.javaDefaults();
        settings.setLanguage(Languages.bytecode());
        File jarFile = new File(this._settings.getInternalName());
        if (!jarFile.exists()) {
        System.out.println(
        "File not found: " + this._settings.getInternalName()
        );
    }
        JarFile jar = new JarFile(jarFile);
        Enumeration<JarEntry> entries = jar.entries();
        settings.setShowSyntheticMembers(false);
        settings.setTypeLoader(new JarTypeLoader(jar));
        this._settings.setSettings(settings);
        _graphContext.setName(jar.getName());
        while (entries.hasMoreElements()) {
                JarEntry entry = entries.nextElement();
                String name = entry.getName();
                if (!name.endsWith(".class")) {
                        continue;
                }
                String internalName;
```

```
                internalName = StringUtilities.removeRight(
                        name, ".class"
                        );
                this._settings.setInternalName(internalName);
                ClassMetadata meta = new ClassMetadata(_settings);
                _graphContext.CreateClassMetadata(
                        meta.getMetadata()
                );
        }
}
```

Metoda `execute()` v první řadě nastaveni vlastnosti dekompilace. Nastavení provádí třída `DecompilerSettings`. Abychom nemuseli nastavovat všechny hodnoty, zavoláme metodu `javaDefaults()`, která nám vrátí standardní nastavení. Nastavení si dále upravíme podle potřeb. Nastavíme jazyk dekompilace metodou `setLanguages()` na hodnotu `Languages.bytecode()`. Následně zkusíme otevřít jar soubor třídou `File`. Cestu k jar souboru máme uloženou v privátní proměnné `_settings.getInternálName()`. Poté ověříme, jestli jsme soubor našli. Pokud máme soubor k dispozici, převedeme ho na objekt `JarFile`. To nám dopomůže k získání jednotlivých souborů, které jsou obsaženy v jar souboru. Třída `JarFile` totiž obsahuje metodu `entries()`, která nám po zavolání vrátí výčtový seznam souborů. Po rozbalení jar souboru můžeme nastavit zbývající vlastnosti pro dekompilaci. Před dekompilací ještě vytvoříme v grafu kořenový vrchol jar souboru.

Následně můžeme začít dekompilovat soubory. Dekompilaci budeme provádět v cyklu. Postupně vybereme všechny soubory, zjistíme název souboru s příponou. Pokud je přípona jiná než class, přeskočíme daný soubor. Dále odstraníme příponu a jméno souboru uložíme do proměnné `internaName`. Tuto proměnou nastavíme v třídě `_settings` metodou `setInternalName()`. Po nastavení můžeme zavolat konstruktor třídy `ClassMethod` (viz sekce5.3.2) s parametrem `_settings`, který provede dekompilaci. Po dekompilaci zavoláme metodu `CreateClassMetadata()` z třídy `GraphContext`. Tato metoda je zavolána s parametrem výsledných metadat z dekompilace, která získáme z metody `getMetadata()` z třídy `ClassMetadata`. Metoda `CreateClassMetadata` uloží metadata do grafové databáze. Tento cyklus opakujeme dokud neprojdeme všechny soubory z jar souboru.

5.3.2 Dekompilace class souboru

O dekompilaci class souborů se stará třída `ClassMetadata`, která využívá nástroj Procyon. Třída `ClassMetadata` rozšiřuje abstraktní třídu `ProcessingData`, která je v balíku `Core`.

```
public ClassMetadata(Setting settings) {
        super(settings);
        Build(settings.getSettings());
}
```

Konstruktor volá děděný konstruktor, který nastaví místo uložení dekompilovaného kódu. Dekompilovaný kód můžeme uložit na místní disk nebo přesměrovat a uložit do

proměnné. Konstruktor dále volá vlastní metodu `Build()` s parametrem, který nastavuje vlastnosti dekompilace.

```
private void Build(final DecompilerSettings settings){
        if(_outputFile == null) {
                try (final OutputStreamWriter writer =
                        new OutputStreamWriter(_outputVar)) {
                        decompile(
                                _internalName,
                                 new PlainTextOutput(writer),
                                  settings
                                 );
                }
                catch (final IOException e) {
                        System.out.println(e);
                }
        }else {
                try (final FileOutputStream stream =
                         new FileOutputStream(_outputFile);
                        final OutputStreamWriter writer =
                         new OutputStreamWriter(stream)
                         ) {
                        decompile(
                                _internalName,
                                 new PlainTextOutput(writer),
                                  settings
                                  );
                }
                catch (final IOException e) {
                System.out.println(e);
                }
        }
}
```

Metoda `Build()` inicializuje výstupní proud, kam data poputují. Jestli se budou ukládat do souboru nebo do proměnné. Poté se zavolá metoda `decompile()` s interním názvem dekompilované třídy, výstupním úložištěm a nastavenými vlastnostmi dekompilace.

Metoda `decompile()` obsahuje převzatý kód z Procyonu určený pro dekompilaci. Navíc obsahuje pár úprav pro získání důležitých dat.

5.4 Implementace databáze

V této sekci popíšeme tvorbu grafové databáze Titan. S databází Titan nepracujeme přímo, ale využíváme framework Tinkerpop. Poté popíšeme prvky databáze a jejich převod na objekty Javy pro pohodlnější použití. K převodu použijeme framework Frame.

5.4.1 Stavba databáze

Třída `GraphContext` slouží pro udržení spojení s databází a její tvorbou. Dále popíšeme její konstruktory a metody[1].

Při zavolání konstruktoru této třídy vytvoříme prázdnou databázi, se kterou udržujeme spojení. Poté vytvoříme továrnu pro převod grafových objektů na objekty Javy a to frameworkem Frames. Přidáme ještě modul `GremlinGroovyModule()` a můžeme vytvořit kontext pro přistup k databázi prostřednictvím Java rozhraní.

Metoda `setName(String name)` slouží pro nastavení počátečního vrcholu, kterým je jar vrchol. Na jar vrchol se poté budou napojovat třídy a balíky. Jako parametr je zadán název jar souboru i s cestou. Vlastnosti vrcholu jsou popsány v kapitole 5.4.2.

Metoda `CreateClassMetadata(TypeDefinition metadata)` vytvoří ze stromu byte kódu podstrom metadat třídy, který napojí na hlavní strom začínající počátečním vrcholem. Při tvorbě databáze nejdříve vyhledáme, jestli třída již neexistuje v databázi. Tento fakt může nastat ve chvíli, kdy třída má nějaký vztah s třídou, která ještě nebyla dekompilovaná. V takovém případě vytvoříme šablonu pro dosud nedekompilovanou třídu. Poté můžeme nastavovat vlastnosti třídy např. jméno, popis, příznaky atd.. Následně se pustíme do nastavení vztahů. Začneme zjištěním zda daná třída je podtřídou některé jiné třídy, v takovém případě vytvoříme vztah `extendsRelated` s nadtřídou. Pokud se nadtřída nachází v jiném balíku, vytvoříme druhy vztah a to `importRelated`, to provede metoda `setImport(String fqn)`. Dále napojíme třídu na hlavní grafovou strukturu a pokračujeme vytvořením případných spojení s rozhraními, které třída implementuje vztahem `implementsRelated`. Nezapomeneme nastavit případný import rozhraní z jiných balíků. Potom zkontrolujeme anotace třídy a případně vytvoříme vrchol pro anotaci a vztah `annotatedRelated`. O nastavení vrcholu se postará metoda `setAnnotation()`. Vlastnosti třídy neukládáme do databáze, ale zajímají nás případné importy objektu z jiných tříd, kde nad typem každé vlastnosti zavoláme metodu `setImport()`. Nakonec zjistíme o jaký typ třídy jde: `class`, `annotation`, `interface` nebo `enum`. U každé třídy nastavíme typ další specifické vlastnosti a vztahy pro daný typ třídy. Pro `class` a `interface` vytvoříme vrcholy pro metody a vztah `methodRelated`. O nastavení vrcholu metody se postará metoda `setMethod()` s parametrem podstromu metody.

Metoda `setAnnotation()` nastaví vlastnosti vrcholu anotace a vytvoří vrcholy pro parametry s propojujícím vztahem `annParaRelated`. Jako parametr anotace může být:

a) anotace – voláme metodu `setAnnotationAnnotationElement()`

b) pole – voláme metodu `setArrayAnnotationElement()`

c) hodnota – voláme metodu `setConstantAnnotationElement()`

d) třída – voláme metodu `setClassAnnotationElement()`

e) výčtový typ – voláme metodu `setEnumAnnotationElement()`

Metoda `setAnnotationAnnotationElement()` nastaví jméno, typ a popis anotace a případně vytvoří vztah `importRelated` pokud se typ anotace nachází v jiném balíku. Dále se nastaví parametry, ty byly popsány v předchozím odstavci.

Metoda `setArrayAnnotationElement()` zjistí, jaké položky má obsahovat a zavolá příslušné metody (viz. `setAnnotation()` nastavení parametrů).

[1] Zdrojový kód třídy je přiložen na CD.

Metoda `setConstantAnnotationElement()` nastaví vrcholu parametru hodnotu parametru.

Metoda `setClassAnnotationElement()` nastaví jako hodnotu parametru název třídy.

Metoda `setEnumAnnotationElement()` nastaví vrcholu hodnotu výčtového typu.

Metoda `setMethod()` nejprve nastaví vlastnosti metody jako je jméno, typ, popis a příznaky. Následně zjistí, jestli se má importovat typ návratové hodnoty, zda má metoda anotaci případně anotace nastaví metodou `setAnnotation()`. Následně vytvoří vrcholy pro parametry a hrany pro vztahy nazvané `MethParaRelated`. U parametru musíme také zjistit případné anotace a nastavit je metodou `setAnnotation()`. Potom spočítáme parametry a metodě nastavíme počet parametrů. A jako poslední věc projdeme tělo funkce, zda tam nejsou volání metod. Volání metod mohou mít pět typů: `INVOKESTATIC`, `INVOKEINTERFACE`, `INVOKEVIRTUAL`, `INVOKEDYNAMIC` a poslední `INVKESPECIAL`. Volání metod nastavuje metoda `setCall()`.

Metoda `setCall()` nastaví případný import volané metody a dále vyhledá volanou metodu. Pokud ji nenajde, vytvoří pro tuto metodu šablonu. Šablony poznáme podle toho, že mají nastavenou vlastnost `NotDecompile` na hodnotu `true`. Po naplnění šablony tuto hodnotu změníme na `false`.

Metoda `getClass()` vyhledá šablonu pro třídu, pokud jí nenajde, vytvoří novu šablonu a vrátí ji.

Metoda `setPKG()` napojí nově vzniklou třídu do databáze na hlavní strom.

Metoda `setImport()` zjistí, jestli přijatou třídu musí importovat nebo ne.

Následující metody `getClassInPackage()` a `getClassInPackageRecursion()` slouží pro získání tříd z balíčků. Metoda `getClassInPackage()` získá pouze třídy z aktuálního balíčku a metoda `getClassInPackageRecursion()` i z podbalíčků.

5.4.2 Prvky databáze

V databázi jsou dva typy prvků, jsou to vrcholy a hrany. Vrcholy jsou objekty databáze, které uchovávají data. Hrany jsou objekty databáze, které propojují tyto daty. V naší databázi jsme však použili framework Frame, kterým jsme převedly vrcholy a hrany na objekty Javy. Pro převod potřebujeme vytvořit rozhraní pro jednotlivé prvky.

Pro vrcholy jsme vytvořili základní rozhraní `BaseEntity`, které obsahuje obecné vlastnosti vrcholu např. jméno, typ vrcholu, plné jméno vrcholu, příznaky a popis. Ostatní vrcholy toto rozhraní dědí a přidávají své další vlastnosti, nejčastěji to jsou vztahy mezi vrcholy.

Prvním nadefinovaným rozhraním je `JarEntity` pro vrchol jar souboru. Toto rozhraní má přidané metody pro správu vztahů vrcholů a to metody `getPackageRelated( @GremlinParam("name") String name)`. Tato metoda vrátí balík zadaného jména a má navíc ještě anotaci `@GremlinGroovy("it.as('x').out('packageRelated').except('x') .has( 'name',name)")`, která specifikuje dotaz do databáze. Další metodou je `getClassRelated(@GremlinParam("name") String name)`, která vrací třídu zadaného jména. Následují metody pro přidání vrcholu tříd `addClassRelated()`, která vrátí nový vrchol pro třídu `addClassRelated(ClassEntity)` a vytvoří vztah už s existující třídou `ClassEntity`. Následuje metoda `getClassRelated()`, která vrátí všechny třídy v jar souboru, které nejsou v balíčku. Další třídy jsou pro vztahy s balíčky. První je `addPackageRelated()`, která přidá nový balíček do jar vrcholu a `addPackageRelated(PackageEntity)` vytvoří vztah s existujícím balíčkem. Poslední metodou je `getPackageRelated()`, která vrátí všechny balíčky v jar vrcholu.

U dalších vrcholů již nebudeme popisovat metody a vlastnosti pouze popíšeme, jaký vrchol reprezentují. Rozhraní `ClassEntity` reprezentuje vrchol pro třídy, `MethodEntity` reprezentuje vrchol pro metody, `MethParaEntity` reprezentuje vrchol pro parametry metody, `AnnotatedEntity` reprezentuje vrchol anotace, `AnnParaEntity` reprezentuje vrchol parametru anotace a `PackageEntity` reprezentuje vrchol pro balíček.

5.5 Interpret jazyka

Interpret jazyka se nachází v balíčku `QueryLanguage`, kde se nacházejí jeho tři podbalíčky. Prvním podbalíčkem je `LexerParser`, ten obsahuje třídy vygenerované třídy nástrojem ANTLR. Druhý podbalíček `SemanticExecute` obsahuje sémantickou analýzu a interpretaci jazyka, což bude náplní této podkapitoly. Třetí balíček `WalkerContext` obsahuje třídy udržující data pro instrukce.

5.5.1 Sémantika a generátor instrukcí

O sémantickou kontrolu a generování instrukcí se stará třída `SemanticGenerator`. Třída `SemanticGenerator` obsahuje metody posluchačů vstupů a výstupů do uzlů, které se spouští při průchodu syntaktickým stromem. Kromě metod posluchačů obsahuje taky třídu `ErrorMessage`, která slouží pro zachycení chybového stavu. Další důležitou součástí této třídy je proměnná `stack` typu `Stack`, do které se ukládají názvy instrukcí a jedna z tříd v balíčku `WalkerContext`, která slouží k uchování dat pro instrukce.

První metodou je `enterSelectStatement()`, tato metoda uloží do proměnné`stack` instrukci `enterSelectStatement` a instanci třídy `SelectStatementContext`, která udržuje data instrukce. Nevadí, že jsme v této metodě neuložili žádná data. Jde nám o to, kde se ve stromu nacházíme.

Metoda `exitSelectStatement()` uloží do proměnné `stack` instrukci `exitSelectStatement` a instanci třídy `SelectStatementContext`. Zase neukládáme data a zajímaná nás pouze opuštění daného uzlu.

Metoda `enterParamSelect()` provádí totéž, pouze uložíme do proměnné `stack` instrukci `enterParamSelect` a instanci třídy `ParamSelectContext`.

Metoda `enterParamName()` provádí totéž, pouze uložíme do proměnné `stack` instrukci `enterParamName` a instanci třídy `ParamNameContext`.

Metoda `exitParamName()` vytvoří instanci třídy `ParamNameContext` a uloží do ní data z listů uzlů, před uložením ještě provede sémantickou kontrolu, zda jsou data v listech uzlů validní. Pokud nastane chyba, vytvoří se zpráva o chybě a je uložena do proměnné `_errMsg`. A proměnná `_error` je nastavena na hodnotu true. V sémantické analýze se však pokračuje dál, ale vykonání instrukcí bude přerušeno, jenom se vypíší chybová hlášení. Na konci se uloží do proměnné `stack` instrukce `exitParamName` a instance třídy `ParamNameContext`.

Metoda `enterPackages()` pouze uloží do proměnné `stack` instrukci `enterPackages` a instanci třídy `PackagesContext`.

Metoda `exitPackages()` pouze uloží do proměnné `stack` instrukci `exitPackages` a instanci třídy `PackagesContext`.

Metoda `enterPackageLink()` pouze uloží do proměnné `stack` instrukci `enterPackageLink` a instanci třídy `PackageLinkContext`.

Metoda `exitPackageLink()` vytvoří instanci třídy `PackageLinkContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci `exitPackageLink` a instanci třídy `PackaheLinkContext`.

Metoda `enterPackageName()` pouze uloží do proměnné `stack` instrukci `enterPackageName` a instanci třídy `PackageNameContext`.

Metoda `exitPackageName()` vytvoří instanci třídy `PackageNameContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `PackageNameContext`.

Metoda `enterConditions()` pouze uloží do proměnné `stack` instrukci `enterConditions` a instanci třídy `ConditionsContext`.

Metoda `enterCond()` pouze uloží do proměnné `stack` instrukci `enterCond` a instanci třídy `CondContext`.

Metoda `exitCond()` vytvoří instanci třídy `CondContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `CondContext`

Metoda `enterEqual()` pouze uloží do proměnné `stack` instrukci `enterEqual` a instanci třídy `EqualContext`.

Metoda `exitEqual()` vytvoří instanci třídy `EqualContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `EqualContext`

Metoda `enterRigthStatement()` pouze uloží do proměnné `stack` instrukci `enterRigthStatement` a instanci třídy `RigthStatementContext`.

Metoda `exitIndex()` vytvoří instanci třídy `IndexContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `IndexContext`

Metoda `exitAs()` vytvoří instanci třídy `AsContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `AsContext`

Metoda `exitAlias()` vytvoří instanci třídy `AliasContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `AliasContext`

Metoda `exitAnnotated()` vytvoří instanci třídy `AnnotatedContext` a uloží do ní data listů daného uzlu. Poté uloží instrukci do proměnné `stack` spolu s vytvořenou instancí třídy `AnnotatedContext`

Metoda `exitInnerSelect()` pouze uloží do proměnné `stack` instrukci `exitInnerSelect` a instanci třídy `InnerSelectContext`.

Následují tři další metody: `PrintErr()` (vypíše sémantické chyby), `isError()` (vrátí true, pokud nastala chyba) a poslední `getStack()` (vrátí seznam instrukcí).

5.5.2 Vyhodnocení instrukcí

O interpretaci instrukcí se stará instance třídy `Stack`, která volá metody z třídy `Interpret` a předává jím třídy obsahující data listů daného uzlu. Vykonání instrukcí započne zavoláním metody `run()` v instanci třídy `Stack`, která postupně vyhodnotí všechny vygenerované instrukce a zavolá metodu z třídy `Interpret`, které předá instance tříd s daty listů uzlu.

Třída `Interpret` má podobné metody jako třída `SemanticGenerator`, které vykonávají jednotlivé instrukce, které třída `SemanticGenerator` vygenerovala.

Kapitola 6

Testování

Tato kapitola se zabývá testováním aplikace. Nejprve uvedeme dvacet testů dotazu do testovacího jar souboru `JavaTestQueryToAST.jar`, který je přiložen k projektu. Poté uvedeme pár příkladu chybových vstupů.

6.1 Testování jazyka

Postup při testování je následující. Položíme dotaz, který v následujícím podstavci popíši. A na konci odstavce vypíši výsledky dotazu. Pokud došlo k chybě vypíši chybová hlášení.

```
SELECT po.*
        FROM (
                SELECT extends, import
                        WHERE name='C'
                ) AS po
                WHERE (
                        SELECT !extends
                                WHERE name = 'I'
                        ) IN po.implements
```

Tento dotaz vybere všechny třídy, které rozšiřují nebo jsou importovány do třídy C. Tyto třídy implementují rozhraní, které je rozšířeno o interface I. Výsledkem je třída `langTest.One.Classes`.

```
SELECT !implements
        WHERE name='I'
```

Tento dotaz vybere všechny třídy implementující rozhraní I. Výsledkem jsou třídy `langTest.One.CcallofHard, langTest.One.CimpHard, langTest.One.imp.supClassB`.

```
SELECT call[*]
        WHERE name='I'
```

Tento dotaz vybere všechny třídy volající metodu z rozhraní I. Výsledkem je třída `langTest.One.CcallofHard`.

```
SELECT call[*]
        WHERE (
                WHERE name='I'
                ) IN extends
```

Tento dotaz vybere všechny třídy volající metodu z rozhraní, které je rozšířeno rozhraním I.

```
SELECT call[name='easy', arg='']
        WHERE (
                WHERE name='I'
                ) IN extends
```

Tento dotaz vybere všechny třídy volající metodu `easy` z rozhraní, které je rozšířeno rozhraním I z danými parametry. Výsledkem je `langTest.One.Classes`.

```
SELECT call[*]
        WHERE @Abasic AND (
                WHERE name='Iext'
                ) IN implements
```

Tento dotaz vybere všechny třídy, které volají metodu z třídy, která implementuje interface Iext a je anotovány anotací `Abasic`. Výsledkem je třída `langTest.One.Classes`.

```
SELECT *
        WHERE @Aauthor.email='jannovak@seznam.cz' AND (
                WHERE name='I'
                ) IN implements
```

Tento dotaz vybere všechny třídy, které volají metodu z třídy, která implementuje interface I a je anotovány anotací `Aauthor(email='jannovak@seznam.cz?)`. Výsledkem je třída `langTest.One.CcallofHard`.

```
SELECT *
        WHERE @Abasic
```

Tento dotaz vybere všechny třídy, které jsou anotované anotací `Abasic`. Výsledkem jsou třídy `langTest.One.CimpOFI, langTest.One.C`.

```
SELECT *
        WHERE (
                SELECT !extends
                WHERE name='supClassB'
                ) IN import
```

Tento dotaz vybere všechny třídy, které importují podtřídu třídy `supClassB`. Výsledkem je třídy `langTest.One.Canot`.

```
SELECT *
        WHERE (
                SELECT !extends
                        WHERE (
                                SELECT !extends
                                        WHERE name='I'
                                ) IN implements
                ) IN import
```

Tento dotaz vybere všechny třídy importující podtřídu třídy implementující pod rozhraní z rozhraní `I`. Výsledkem je třída `langTest.One.Canot`.

```
SELECT za.*
        FROM (
                        WHERE name='Protokol'
                ) AS po, * AS za
                WHERE za.name=po.@Dependencies.value[0].
                @Dependency.value
```

Tento dotaz vybere všechny třídy, na kterých je třída `Protokol` závislá vzhledem k anotaci třídy `Protokol` prvního parametru.

```
SELECT (
        SELECT (
                SELECT(
                        SELECT extends
                        )))
```

Tento dotaz vybere všechny super třídy. Výsledkem jsou třídy `langTest.One.I`, `langTest.One.imp.supClassB`, `langTest.One.Classes`, `test.AbstractTest`, `test.AbstractTest`. Tento dotaz tu uvádím jako ukázku několikanásobného zanoření.

```
SELECT *
```

```
        WHERE name != 'C'
```

Tento dotaz vybere všechny třídy, které se nejmenují C. Výsledek jsou všechny třídy kromě třídy C.

```
SELECT ap.*
        FROM (SELECT implements) AS ap,
                (SELECT extends) join (SELECT import) AS ExIm
        WHERE ap.name=ExIm.name
```

Tento dotaz vybere všechna rozhraní, která jsou někde importovaná. Tento dotaz hlavně testuje spojení dvou množin tříd.

```
SELECT ps.*
        FROM (
                        WHERE name=r '.*In.*'
                ) AS ps
                WHERE exist (
                        SELECT *
                                WHERE exist (
                                        SELECT *
                                                WHERE ps.name=name
                                                ))
```

Tento dotaz testuje zanoření aliasů do vnitřních dotazů a regulární výraz v dotazu. Výsledkem je `anotTest.Inject, used.InterfaceUsedTest, used.AbsInterfUsedTest, test.InterfaceTest, test.InterfaceTest2, test.ClassInClass.first.second, test.ClassInClass, test.ClassInClass.first`.

```
SELECT ps.*, pe.*
        FROM * AS ps, * AS pe
                WHERE ps.inner AND pe.interface
```

Tento dotaz vyhledá vnitřní třídy a rozhraní a vypíše je. Výsledkem jsou třídy `test.ClassInClass.first.second, test.ClassInClass.first, langTest.One.I, langTest.One.Iext, test.InterfaceTest, test.InterfaceTest2`.

```
SELECT *
        WHERE @Retention AND @Target
```

Tento dotaz vyhledá všechny třídy anotované anotacemi `Retention` a `Target`. Výsledkem jsou třídy `langTest.One.Aauthor, langTest.One.Abasic, anotTest.Dependency,`

`anotTest.Security, anotTest.Constructor, anotTest.Dependencies, anotTest.Readonly, anotTest.Inject`.

```
SELECT call[name='getName']
        FROM 'test'
```

Tento dotaz vyhledá třídy volající metodu `getName`. Výsledkem jsou třídy `used.AbstractUsedTest, test.NormalTest`.

```
SELECT extends
        FROM !'langTest.One'
```

Tento dotaz vyhledá supertřídy a to pouze v balíku `langTest.One`, ale v podbalících nevyhledává. Výsledkem dotazu je `langTest.One.I`.

```
SELECT extends
        FROM 'langTest.One'
```

Tento dotaz vyhledá supertřídy a to i v podbalících balíku `langTest.One`. Výsledkem dotazu je `langTest.One.I, langTest.One.imp.supClassB, langTest.One.Classes`.

```
SELECT extend
        WHERE name='C' AND nterface
```

Tento dotaz vypíše dvě chyby `Error: Neznámí parametr:extend` a `Error: Neznáme klíčové slovo:nterface`.

```
SELECT po.extends
        FROM * AS pe
                WHERE po.name='C'
```

Tento dotaz je také chybný alias `po` nebyl definován. Výsledkem je chyba `Error: Neexistující alias: po`.

```
SELECT po.extends
        FROM * AS pe
                WHERE exist (
                        SELECT pe.extends
                                FROM * AS po
                        ) AND po.name='C'
```

Tento dotaz je chybný také, alias je sice definován, ale ve vnitřním dotazu takže hlavní dotaz ho nevidí. Výsledkem je `Error: Neexistující alias: po`.

```
SELECT call[name='getName', arg='int,String', name='setName']
        FROM *
                WHERE name='NormalTest'
```

Tento dotaz je chybný, protože má duplicitní argumenty ve volání metody a má 3 parametry, avšak povolené jsou maximálně 2 parametry. Výsledkem je `Error: Maximální počet argumentů je 2 ne:3` a `Error: Duplicitní parametr:name`.

6.2 Výsledky testování

Aplikace dokáže zpracovat celou řadu nejrůznějších dotazů. Pomocí vnořování dotazů do sebe a použitím aliasů je možné vytvořit složité dotazy. Tam si už musíme dávat pozor, abychom neudělali nějakou chybu. Aplikace bude většinou sloužit pro pokládání jednodušších dotazů. Nejčastěji se budeme dotazovat na názvy tříd. Aplikace má v sobě zabudovanou testovací třídu pro rychlý test stability aplikace a funkčnosti jazyka. K jejímu spuštění musíme zadat cestu k testovacímu jar souboru `JavaTestQueryToAST.jar`, který je v příloze na CD.

Kapitola 7

Závěr

V tomto projektu jsem se hodně naučil. Například práci s několika nástroji jmenovitě: Procyon (dekompilátor jazyka Java), Titan (grafová databáze) a její framework TinkerPop a nástroj ANTLR (generátor překladačů). Dále jsem si prohloubil znalosti ohledně samotného jazyka Java, hlavně ohledně JVM.

Jako vlastní přínos mohu jmenovat vytvoření jazyka k vyhledávání v grafové databázi, který se specializuje na vyhledávání tříd podle zadaných kritérií. Jeho použití je určeno pro analýzu class souborů jazyka Java.

Další vývoj vidím především v optimalizaci kladení dotazu, rozšíření klíčových slov jazyka pro rozšíření funkčnosti, přidání agregačních funkcí a odstranění nenalezených chyb. Optimalizaci kladení dotazů mohu provést vytvořením indexu pro třídy, balíky, metody a anotace. Dále lze vytvořit binární vyhledávací strom pro tyto položky. Díky této optimalizaci nebudeme muset začínat vyhledávání od kořene stromu, ale můžeme začít vyhledávání uvnitř stromu. Dále bychom mohly rozšířit jazyk o možnost, aby výsledkem mohl být balík, metoda, parametr nebo anotace. Výsledkem budou vlastnosti uzlů místo ukazatelů na uzly. Z toho plyne také možnost řadit výsledky podle vlastností uzlů. Další možností rozšíření jazyka je přidání agregačních funkcí. Jsou to počet výsledků, minimální hodnota, maximální hodnota, průměrná hodnota, součet atd. Na závěr bychom mohli rozšířit jazyk o dotazování do byte kódu a zdrojového kódu jazyka Java.

Literatura

[1] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: The Java Virtual Machine Specification: Java SE 8 Edition. `https://docs.oracle.com/javase/specs/index.html`, March 2015, jSR-337.

[2] Mallette, S.: Frames. An Object to Graph Framework [online]. `https://github.com/tinkerpop/frames/wiki`, 2014 [cit. 2015-3-30].

[3] Mallette, S.: Rexster. A Graph Server [online]. `https://github.com/tinkerpop/rexster/wiki`, 2014 [cit. 2015-3-30].

[4] Mallette, S.: Blueprints. A Property Graph Model Interface [online]. `https://github.com/tinkerpop/blueprints/wiki`, 2015 [cit. 2015-3-30].

[5] Parr, T.: *The Definitive ANTLR 4 Reference. 2. vyd.* US: Pragmatic Bookshelf, 2013, iSBN 978-1934356999.

[6] Parr, T.: ANTLR. ANother Tool for Language Recognition [online]. `http://www.antlr.org/`, 2015 [cit. 2015-3-30].

[7] Rodriguez, M. A.: TinkerPop. A Graph Computing Framework [online]. `https://github.com/tinkerpop`, 2015 [cit. 2015-3-30].

[8] Shinavier, J.: Gremlin. A Graph Traversal Language [online]. `https://github.com/tinkerpop/gremlin/wiki`, 2014 [cit. 2015-3-30].

[9] Strobel, M.: Procyon. Java Decompilator [online]. `https://bitbucket.org/mstrobel/procyon/wiki/Home`, 2015 [cit. 2015-3-30].

Příloha A

Obsah CD

Java-AST-query-language ..... adresář se zdrojovým kódem
queryToAST.jar ..... spustitelná aplikace
Manual.pdf ..... manuál pro práci s aplikací
projekt.pdf ..... text práce ve formátu pdf
projekt ..... zdrojový formát práce v LaTeX
JavaTestQueryToAST.jar ..... testovací jar soubor